

*Comportamiento  
y  
lógica de  
negocio*

lección 8

OpenXava no es simplemente un marco de trabajo para hacer mantenimientos (altas, bajas, modificaciones y consultas), más bien está concebido para desarrollar aplicaciones de gestión plenamente funcionales. Hasta ahora hemos aprendido como crear y refinar la aplicación para manejar los datos. Ahora vamos a posibilitar al usuario la ejecución de lógica de negocio específica.

En esta lección vamos a ver como escribir lógica de negocio en el modelo y llamar a esta lógica desde acciones personalizadas. Así podrás transformar tu aplicación de gestión de datos en una herramienta útil para el trabajo cotidiano de tu usuario.

## 8.1 Lógica de negocio desde el modo detalle

Empezaremos con el caso más simple: un botón en modo detalle para ejecutar cierta lógica. En este caso para crear la factura desde un pedido (figura 8.1).

Estamos en el módulo Order

1. El usuario escoge un pedido

2. Entonces pulsa en el botón 'Create invoice'

3. Se muestra un mensaje confirmando la creación de la nueva factura

4. Se muestra la nueva factura en la pestaña 'Invoice' del pedido

Las líneas de detalle de la nueva factura han sido copiadas del pedido actual

	Number of Product	Description of Product	Quantity	Price per unit	Amount
	1	Peopleware: Productive Projects and Teams	2	31.00	62.00
	2	Arco iris de lágrimas	1	15.00	15.00

Figura 8.1 Crear una factura desde un pedido usando una acción

## 159 Lección 8: Comportamiento y lógica de negocio

La figura 8.1 muestra como esta nueva acción coge el pedido actual y crea una factura a partir de él. Simplemente copia todos los datos del pedido a la nueva factura, incluyendo las líneas de detalle. Se muestra un mensaje y la pestaña 'Factura' del pedido visualizará la factura recién creada. Veamos como codificar este comportamiento.

### 8.1.1 Crear una acción para ejecutar lógica personalizada

Como ya sabes el primer paso para tener una acción personalizada en tu módulo es definir un controlador con esa acción. Por tanto, editemos *controllers.xml* y añadamos un nuevo controlador. El listado 8.1 muestra el controlador Order.

**Listado 8.1 Controlador Order en controllers.xml, con la acción createInvoice**

```
<controller name="Order">
  <extends controller="Invoicing"/> <!-- Para tener las acciones estándar -->

  <action name="createInvoice" mode="detail"
    class="org.openxava.invoicing.actions.CreateInvoiceFromOrderAction"/>
  <!-- mode="detail" : Sólo en modo detalle -->

</controller>
```

Dado que hemos seguido la convención de dar al controlador el mismo nombre que a la entidad y el módulo, ya tenemos automáticamente esta nueva acción disponible para Order. El controlador Order descende del controlador Invoicing. Recuerda que creamos un controlador Invoicing en la lección 7. Es un refinamiento del controlador Typical.

Ahora hemos de escribir el código Java para la acción. Puedes verlo en el listado 8.2.

**Listado 8.2 Código de la acción para crear una factura desde un pedido**

```
package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.jpa.*;

public class CreateInvoiceFromOrderAction
  extends ViewBaseAction { // Para usar getView()

  public void execute() throws Exception {
    Order order = XPersistence.getManager().find( // Usamos JPA para obtener la
      Order.class,                               // entidad Order visualizada en la vista
      getView().getValue("oid"));
    order.createInvoice(); // El trabajo de verdad lo delegamos en la entidad
    getView().refresh();  // Para ver la factura creada en la pestaña 'Invoice'
    addMessage("invoice_created_from_order", // Mensaje de confirmación
      order.getInvoice());
  }
}
```

```
}
```

Realmente simple. Buscamos la entidad `Order`, llamamos al método `createInvoice()`, refrescamos la vista y mostramos un mensaje. Nota como la acción es un mero intermediario entre la vista (la interfaz de usuario) y el modelo (la lógica de negocio).

Recuerda añadir el texto del mensaje en el archivo *Invoicing-messages\_en.properties* de la carpeta *il8n*. El listado 8.3 muestra un posible texto.

#### Listado 8.3 Mensaje de confirmación en *Invoicing-messages\_en.properties*

```
invoice_created_from_order=Invoice {0} created from current order
```

Sin embargo, el mensaje tal cual está no se muestra de forma agradable, porque enviamos como argumento un objeto `Invoice`. Necesitamos un `toString()` para `Invoice` y `Order` que sea útil para el usuario. Sobrescribiremos `toString()` de `CommercialDocument` (el padre de `Invoice` y `Order`) para conseguirlo. Puedes ver este método `toString()` en el listado 8.4.

#### Listado 8.4 Método `toString()` de `CommercialDocument`

```
abstract public class CommercialDocument extends Deletable {
    ...
    public String toString() {
        return year + "/" + number;
    }
}
```

El año y el número son perfectos para identificar una factura o pedido desde el punto de vista del usuario.

Esto es todo para la acción. Veamos la pieza restante. El método `createInvoice()` de la entidad `Order`.

### 8.1.2 Escribiendo la lógica de negocio real en la entidad

La lógica de negocio para crear una nueva `Invoice` está en la entidad `Order`, no en la acción. Esto es la forma natural de hacerlo. El principio esencial de la Orientación a Objetos es que los objetos no son solo datos, sino datos y lógica. El código más bello es aquel cuyos objetos contienen la lógica para manejar sus propios datos. Si tus entidades son meros contenedores de datos (simples envoltorios de las tablas de la base de datos) y tus acciones tienen toda la lógica para manipularlos, en ese caso tu código es una perversión del objetivo original

de la Orientación a Objetos<sup>11</sup>.

Aparte de las razones espirituales, poner la lógica para crear una Invoice dentro de Order es un enfoque pragmático, porque de esta forma podemos usar esta lógica desde otras acciones, proceso masivos, servicios web, etc.

Veamos el código. El listado 8.5 muestra el método createInvoice() de la clase Order.

### Listado 8.5 Método createInvoice() en la entidad Order

```
public class Order extends CommercialDocument {  
    ...  
    public void createInvoice() throws Exception { // throws Exception para tener  
                                                    // un código más simple, de momento  
        Invoice invoice = new Invoice(); // Instancia una factura  
        BeanUtils.copyProperties(invoice, this); // y copia el estado del pedido actual  
        invoice.setOid(null); // Para que JPA sepa que esta entidad todavía no existe  
        invoice.setDate(new Date());  
        invoice.setDetails(new ArrayList()); // Borra la colección de detalles  
        XPersistence.getManager().persist(invoice);  
        copyDetailsToInvoice(invoice); // Rellena la colección de detalles  
        this.invoice = invoice; // Siempre después de persist()  
    }  
}
```

La lógica consiste en crear un nuevo objeto Invoice, copiar los datos desde el Order actual a él y asignar la entidad resultante a la referencia invoice del Order actual.

Hay dos sutiles detalles aquí. Primero, has de escribir `invoice.setOid(null)`, si no la nueva Invoice tendría la misma identidad que el Order original, además a JPA no le gusta persistir los objetos con el id autogenerado relleno de antemano. Segundo, has de asignar la nueva Invoice a la actual Order (`this.invoice = invoice`) después de llamar a `persist(invoice)`, si no obtendrás un error de JPA (algo así como “object references an unsaved transient instance”).

### 8.1.3 Escribe menos código usando Apache Commons BeanUtils

Observa como hemos usado `BeanUtils.copyProperties()` para copiar todas las propiedades del actual Order a la nueva Invoice. Este método copia todas las propiedades con el mismo nombre de un objeto a otro, incluso si los objetos son de clases diferentes. Esta utilidad pertenece al proyecto de apache Commons BeanUtils. El jar para esta utilidad, *commons-beanutils.jar*, ya está incluido en tu proyecto.

---

<sup>11</sup> Por desgracia muchos de los patrones y buenas prácticas J2EE son perversiones de la Orientación a Objetos

El listado 8.6 muestra como usando BeanUtils escribes menos código.

#### Listado 8.6 BeansUtil.copyProperties() frente a copiar las propiedades a mano

```
BeanUtils.copyProperties(invoice, this);
// Es lo mismo que
invoice.setOid(getOid());
invoice.setYear(getYear());
invoice.setNumber(getNumber());
invoice.setDate(getDate());
invoice.setDeleted(isDeleted());
invoice.setCustomer(getCustomer());
invoice.setVatPercentage(getVatPercentage());
invoice.setAmount(getAmount());
invoice.setRemarks(getRemarks());
invoice.setDetails(getDetails());
```

Sin embargo, la principal ventaja de usar BeanUtils no es ahorrar tiempo de tecleo, sino que obtienes un código más resistente a los cambios. Porque, si añades, quitas o renombtras alguna propiedad de ComercialDocument (el padre de Invoice y Order), si estás copiando las propiedades a mano tienes que cambiar el código, mientras que si estás usando BeanUtils.copyProperties() el código funcionará siempre bien, sin tener que cambiarlo.

### 8.1.4 Copiar una colección de entidad a entidad

La nueva Invoice tiene que tener las mismas líneas de detalle que el Order. Realmente, no la misma colección sino una copia. No podemos asignar la colección tal como muestra el listado 8.7.

#### Listado 8.7 Manera incorrecta de copiar una colección de una entidad a otra

```
invoice.setDetails(getDetails()); // Esto no funciona
```

Esto no funciona porque una misma colección uno-a-muchos no se puede asignar a dos entidades al mismo tiempo, por tanto hemos de hacer una copia. Nota como en el método createInvoice() (listado 8.4) usamos invoice.setDetails(new ArrayList()) para reiniciar la colección. Esto es porque BeanUtils.copyProperties() ha copiado la colección details de Order. De hecho, copia todo aquello que tenga *getter* y *setter*.

El listado 8.8 muestra el método copyDetailsToInvoice() que copia la colección details de Order a Invoice.

#### Listado 8.8 Método copyDetailsToInvoice() en la entidad Order

```
private void copyDetailsToInvoice(Invoice invoice) throws Exception {
    for (Detail orderDetail: getDetails()) { // Itera por los detalles del pedido actual
        Detail invoiceDetail = (Detail) // Clona el detalle (1)
            BeanUtils.cloneBean(orderDetail);
        invoiceDetail.setOid(null); // Para ser grabada como una nueva entidad (2)
        invoiceDetail.setParent(invoice); // El punto clave: poner un nuevo padre (3)
    }
}
```

```

        XPersistence.getManager().persist(invoiceDetail); // (4)
    }
}

```

Esta es la forma más simple de clonar una colección, simplemente clonando cada elemento (1) y asignándole un nuevo padre (3). También has de quitarle su identidad (2) y marcarlo como persistente (4).

Para clonar el bean usamos BeanUtils otra vez, en este caso el método cloneBean(). Este método crea una nueva instancia del mismo tipo que el argumento, y después copia todas las propiedades del objeto fuente en el objeto recién creado.

### 8.1.5 Excepciones de aplicación

Recuerda la frase: “La excepción que confirma la regla”. Las reglas, la vida, y el software están llenos de excepciones. Y nuestro método createInvoice() no es una excepción. Hemos escrito código que funciona en los casos más comunes. Pero, ¿qué ocurre si el pedido no está listo para ser facturado, o si hay algún problema para acceder a la base de datos? Obviamente, en este caso necesitamos tomar caminos diferentes.

Es decir, el simple throws Exception que hemos escrito para el método createInvoice() no es suficiente para un comportamiento refinado. El listado 8.9 es una versión mejorada del método, usando excepciones.

#### Listado 8.9 El método createInvoice() manejando casos excepcionales

```

public void createInvoice()
    throws ValidationException // Una excepción de aplicación (1)
{
    if (this.invoice != null) { // Si ya tiene una factura no podemos crearla
        throw new ValidationException( // Admite un id de 18n como argumento
            "impossible_create_invoice_order_already_has_one");
    }
    if (!isDelivered()) { // Si el pedido no está entregado no podemos crear la factura
        throw new ValidationException(
            "impossible_create_invoice_order_is_not_delivered");
    }
    try {
        Invoice invoice = new Invoice();
        BeanUtils.copyProperties(invoice, this);
        invoice.set0id(null);
        invoice.setDate(new Date());
        invoice.setDetails(new ArrayList());
        XPersistence.getManager().persist(invoice);
        copyDetailsToInvoice(invoice);
        this.invoice = invoice;
    }
    catch (Exception ex) { // Cualquier excepción inesperada (2)
        throw new SystemException( // Se lanza una excepción runtime (3)
            "impossible_create_invoice", ex);
    }
}

```

```
}
}
```

Ahora declaramos explícitamente las excepciones de aplicación que este método lanza (1). Una excepción de aplicación es una excepción chequeada que indica un comportamiento especial pero esperado del método. Una excepción de aplicación está relacionada con la lógica de negocio del método. Puedes crear una excepción de aplicación para cada posible caso. Por ejemplo, podrías crear una `OrderAlreadyHasInvoiceException` y una `InvoiceNotDeliveryException`. Esto te permitiría tratar cada caso de forma diferente desde el código que usa el método. Aunque, esto no es necesario en nuestro caso, por tanto nosotros simplemente usamos `ValidationException`, una excepción de aplicación genérica incluida con OpenXava.

También hemos de enfrentarnos a problemas inesperados (2). Los problemas inesperados incluyen errores del sistema (acceso a base de datos, la red o problemas de hardware) o errores de programación (`NullPointerException`, `IndexOutOfBoundsException`, etc). Cuando nos encontramos con cualquier problema inesperado lanzamos una `RuntimeException`. En este caso hemos lanzado una `SystemException`, una `RuntimeException` incluida en OpenXava por comodidad, pero puedes lanzar la `RuntimeException` que quieras.

No necesitas modificar el código de la acción. Si tu acción no atrapa las excepciones, OpenXava lo hace automáticamente. Muestra los mensajes de las `ValidationExceptions` al usuario; y para las excepciones *runtime*, muestra un mensaje de error genérico y aborta la transacción.

Para rematar, añadimos el mensaje para la excepción en los archivos `i18n`. Edita el archivo `Invoicing-messages_en.properties` de la carpeta `Invoicing/i18n` añadiendo las entradas del listado 8.10.

#### Listado 8.10 Mensajes usados por las excepciones

```
impossible_create_invoice_order_already_has_one=Impossible to create invoice:
the order already has an invoice
impossible_create_invoice_order_is_not_delivered=Impossible to create invoice:
the order is not delivered yet
impossible_create_invoice=Impossible to create invoice
```

Hay cierto debate en la comunidad de desarrolladores sobre la manera correcta de usar las excepciones en Java. El enfoque usado en esta sección es la forma clásica de trabajar con excepciones en el mundo J2EE.

### 8.1.6 Validar desde la acción

Usualmente el mejor lugar para las validaciones es el modelo, es decir, las



## 165 Lección 8: Comportamiento y lógica de negocio

entidades. Sin embargo, a veces es necesario poner lógica de validación en las acciones. Por ejemplo, si quieres preguntar por el estado actual de la interfaz gráfica has de hacer la validación en la acción.

En nuestro caso si el usuario pulsa en “Crear factura” cuando está creando un nuevo pedido que todavía no ha grabado, fallará. Falla porque es imposible crear una factura desde un pedido inexistente. El usuario ha de grabar el pedido primero.

Modificamos el método `execute()` de `CreateInvoiceFromOrderAction` para validar que la factura visualizada actualmente esté grabada (listado 8.11).

### Listado 8.11 Validación desde la acción para preguntar por el estado de la vista

```
public void execute() throws Exception {
    Object oid = getView().getValue("oid");
    if (oid == null) { // Si el oid es nulo el pedido actual no se ha grabado todavía
        addError(
            "impossible_create_invoice_order_not_exist");
        return;
    }
    MapFacade.setValues("Order", // Si el pedido existe lo grabamos (2)
        getView().getKeyValues(), getView().getValues());
    Order order = getManager().find(
        Order.class, oid);
    order.createInvoice();
    getView().refresh();
    addMessage("invoice_created_from_order",
        order.getInvoice());
}
```

La validación consiste en verificar que el `oid` es nulo (1), en cuyo caso el usuario está introduciendo un pedido nuevo, pero todavía no lo ha grabado. En este caso se muestra un mensaje y se aborta la creación de la factura. Si el pedido ya existe grabamos los datos desde la interfaz de usuario a la base de datos usando `MapFacade` (2). Es importante tener la base de datos sincronizada con la vista antes de llamar al método de la entidad para crear la factura. Imagina que el usuario marca el pedido como entregado (*delivered*) y después pulsa en “Create invoice”. En este caso obtendría un mensaje de error “Pedido no entregado”. Esto puede ser confuso, por tanto grabar la entidad automáticamente antes de llamar a un método de la entidad es buena idea. Fíjate como `MapFacade` es una herramienta muy útil para mover datos de la interfaz de usuario al modelo.

Aquí también tenemos un mensaje para añadir al archivo `i18n`. Edita el archivo `Invoicing-messages_en.properties` de la carpeta `Invoicing/i18n` añadiendo la entrada mostrada en el listado 8.12.

### Listado 8.12 Mensaje usado por la validación de la acción

```
impossible_create_invoice_order_not_exist=Impossible to create invoice: the
order does not exist yet
```

Las validaciones le dicen al usuario que ha hecho algo mal. Esto es necesario, por supuesto, pero es mejor aún crear una aplicación que ayude al usuario a evitar hacer las cosas mal. Veamos una forma de hacerlo en la siguiente sección.

### 8.1.7 Evento *OnChange* para ocultar/mostrar una acción por código

Nuestro actual código es suficientemente robusto como para prevenir que equivocaciones del usuario estropeen los datos. Vamos a ir un paso más allá, impidiendo que el usuario se equivoque. Ocultaremos la acción para crear una nueva factura cuando el pedido no esté listo para ello.

OpenXava permite ocultar y mostrar acciones automáticamente. También permite ejecutar una acción cuando cierta propiedad sea cambiada por el usuario en la interfaz de usuario. Con estos dos ingredientes podemos mostrar el botón sólo cuando la acción esté lista para ser usada.

Recuerda que una factura puede ser generada desde un pedido si el pedido ha sido entregado y no tiene factura todavía. Por tanto, tenemos que vigilar los cambios en la referencia *invoice* y la propiedad *delivered* de la entidad *Order*. Haremos esto usando la anotación *@OnChange* como se muestra en el listado 8.13.

**Listado 8.13 @OnChange añadido a invoice y delivered en Order**

```
public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange>ShowHideCreateInvoiceAction.class
    private Invoice invoice;

    @OnChange>ShowHideCreateInvoiceAction.class
    private boolean delivered;

    ...
}
```

Con el código de arriba cuando el usuario cambia el valor de *delivered* o *invoice* en la pantalla, la acción *ShowHideCreateInvoiceAction* se ejecutará. Observa el código de la acción en el listado 8.14.

**Listado 8.14 Acción para mostrar/ocultar la acción createInvoice dinámicamente**

```
package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Necesario para usar OnChangePropertyAction,
                                // IShowActionAction and IHideActionAction
public class ShowHideCreateInvoiceAction
    extends OnChangePropertyBaseAction // Necesario para acciones @OnChange (1)
    implements IShowActionAction, // Para mostrar una acción
               IHideActionAction { // Para ocultar una acción
```

```

private boolean show; // Si true la acción 'Order.createInvoice' se mostrará

public void execute() throws Exception {
    show = isOrderCreated() // Establecemos el valor de 'show'. Este valor
        && isDelivered() // se usará en los métodos de abajo:
        && !hasInvoice(); // getActionToShow() y getActionToHide() (2)
}

private boolean isOrderCreated() {
    return getView().getValue("oid") != null; // Leemos el valor desde la vista
}

private boolean isDelivered() {
    Boolean delivered = (Boolean)
        getView().getValue("delivered"); // Leemos el valor desde la vista
    return delivered == null?false:delivered;
}

private boolean hasInvoice() {
    return getView().getValue("invoice.oid") != null; // Leemos el valor
                                                    // desde la vista
}

public String getActionToShow() { // Obligatorio por causa de IShowActionAction
    return show?"Order.createInvoice":""; // La acción a mostrar (3)
}

public String getActionToHide() { // Obligatorio por causa de IHideActionAction
    return !show?"Order.createInvoice":""; // La acción a ocultar (3)
}
}

```

Ésta es una acción convencional con un método `execute()`, aunque extiende de `OnChangePropertyBaseAction` (1). Todas las acciones anotadas con `@OnChange` tienen que implementar `IONChangePropertyAction`, aunque es más fácil extender de `OnChangePropertyBaseAction` la cual lo implementa. Desde esta acción puedes usar `getNewValue()` y `getChangedProperty()`, aunque en este caso concreto no los necesitamos.

El método `execute()` pone a true el campo `show` si la orden visualizada está grabada, entregada y no tiene factura (2). Este campo `show` se usa en los métodos `getActionToShow()` y `getActionToHide()`. Estos métodos indican el nombre calificado de la acción a ocultar o mostrar (3). Así, ocultamos o mostramos la acción `Order.createInvoice`, mostrándola solo cuando proceda.

Ahora puedes probar el módulo `Order`. Verás como cuando marcas o desmarcas la casilla entregado (*delivered*) o escoges una factura, el botón para la acción se muestra u oculta. También, cuando el usuario pulsa en 'Nuevo' para crear un nuevo pedido el botón para crear la factura se oculta. Sin embargo, al editar un pedido ya existente, el botón estará siempre presente, aunque el pedido no cumpla los requisitos. Esto es porque cuando un objeto se busca y visualiza las acciones `@OnChange` no se ejecutan por defecto. Podemos cambiar esto con una

pequeña modificación en SearchExcludingDeleteAction. Miralo en el listado 8.15.

#### Listado 8.15 La acción de búsqueda extiende SearchExecutingOnChangeAction

```
public class SearchExcludingDeletedAction
    extends SearchByKeyAction {
    extends SearchExecutingOnChangeAction { // Usa ésta como clase base
```

La acción de búsqueda por defecto, es decir, SearchByKeyAction no ejecuta las acciones @OnChange por defecto, por tanto cambiamos nuestra acción de buscar para que extienda de SearchExecutingOnChangeAction. SearchExecutingOnChangeAction se comporta exactamente igual que SearchByKeyAction pero ejecutando los eventos *OnChange*. De esta forma cuando el usuario escoge un pedido la acción ShowHideCreateInvoiceAction se ejecuta.

Nos queda un pequeño detalle para que todo esto sea perfecto: cuando el usuario pulsa en 'Crear factura' después de que la factura se haya creado el botón se tiene que ocultar. El usuario no puede crear la factura otra vez. Podemos implementar esta funcionalidad con un ligero refinamiento de CreateInvoiceFromOrderAction. Veámoslo en el listado 8.16.

#### Listado 8.16 CreateInvoiceFromOrderAction se oculta a sí misma

```
public class CreateInvoiceFromOrderAction extends ViewBaseAction
    implements IHideActionAction // Para ocultar la acción
{
    private boolean hideAction = false; // Para indicar si la acción se ocultará

    public void execute() throws Exception {
        ...
        addMessage("invoice_created_from_order",
            order.getInvoice());
        hideAction = true; // Todo ha funciona a la perfección, así que ocultamos la acción
    }

    public String getActionToHide() { // La acción a ocultar, en este caso ella misma
        return hideAction?"Order.createInvoice":null;
    }
}
```

Como puedes ver la acción implementa IHideActionAction para ocultarse a sí misma.

Mostrar y ocultar acciones no es un sustituto para la validación en el modelo. Las validaciones siguen siendo necesarias porque las entidades pueden ser usadas desde cualquier otra parte de la aplicación, no solo de los módulos de mantenimiento. Sin embargo, el truco de ocultar y mostrar acciones mejora la experiencia del usuario.

## 8.2 Lógica de negocio desde el modo lista

En la lección 7 aprendiste como crear acciones de lista. Las acciones de lista son una herramienta utilísima para dar al usuario la posibilidad de aplicar lógica a varios objetos a la vez. En nuestro caso, podemos añadir una acción en el modo lista para crear una nueva factura automáticamente a partir de varios pedidos seleccionados en la lista. La figura 8.2 muestra la forma en que queremos que esta acción funcione.

The screenshot shows the 'Invoicing - Order' interface. At the top, a green message box states: 'Invoice 2010/6 created from orders [2010/4, 2010/3]'. Below this is a table of orders. The table has columns: Year, Number, Date, Amount, and Ren. The first row is a header with dropdown menus for each column. The second row is a separator with a green play button icon. The third row has a checkbox, Year: 2010, Number: 1, Date: 2/15/10, Amount: 232.00, and Ren: start. The fourth row has a checkbox, Year: 2010, Number: 2, Date: 2/15/10, Amount: 22.04, and Ren: start. The fifth row has a checked checkbox, Year: 2010, Number: 4, Date: 2/16/11, Amount: 185.60, and Ren: start. The sixth row has a checked checkbox, Year: 2010, Number: 3, Date: 2/15/10, Amount: 23.20, and Ren: start. At the bottom, there are two buttons: 'Delete selected' and 'Create invoice from selected orders'. Three numbered arrows point to specific elements: 1. Points to the checked checkboxes in the 'Number' column of the fifth and sixth rows. 2. Points to the 'Create invoice from selected orders' button. 3. Points to the green message box at the top.

	Year	Number	Date	Amount	Ren
	=	=	=	=	start
<input type="checkbox"/>	2010	1	2/15/10	232.00	start
<input type="checkbox"/>	2010	2	2/15/10	22.04	start
<input checked="" type="checkbox"/>	2010	4	2/16/11	185.60	start
<input checked="" type="checkbox"/>	2010	3	2/15/10	23.20	start

1. El usuario marca los pedidos a facturar

2. Entonces pulsa en el botón "Create invoice..."

3. La factura se crea y se muestra un mensaje

1

Delete selected Create invoice from selected orders

**Figura 8.2 Crear una factura desde varios pedidos usando una acción de lista**

La figura 8.2 muestra como esta acción de lista coge los pedidos seleccionados y crea una factura a partir de ellos. Simplemente copia los datos del pedido en la nueva factura, añadiendo las línea de detalle de todos los pedidos en una única factura. También se muestra un mensaje. Veamos como codificar este comportamiento.

### 8.2.1 Acción de lista con lógica propia

Como ya sabes, el primer paso para tener una acción propia en tu módulo es añadirla a un controlador. Por tanto, editemos *controllers.xml* añadiendo una nueva acción al controlador Order. El listado 8.17 muestra el controlador Order modificado.

**Listado 8.17 Controlador Order con la acción createInvoiceFromSelectedOrders**

```

<controller name="Order">
  <extends controller="Invoicing"/>

  <action name="createInvoice" mode="detail"
    class=
      "org.openxava.invoicing.actions.CreateInvoiceFromOrderAction">
    <use-object name="xava_view"/>
  </action>

  <!-- La nueva acción -->
  <action name="createInvoiceFromSelectedOrders"
    mode="list"
    class=
      "org.openxava.invoicing.actions.CreateInvoiceFromSelectedOrdersAction"
  />
  <!-- mode="list" Solo mostrada en modo lista -->

</controller>

```

Solo con esto ya tienes una nueva acción disponible para Order en modo lista.

Ahora hemos de escribir el código Java para la acción. Míralo en el listado 8.18.

#### Listado 8.18 Acción de lista que crea una factura a partir de varios pedidos

```

public class CreateInvoiceFromSelectedOrdersAction
  extends TabBaseAction // Típico para acciones de lista. Permite usar getTab() (1)
{
  public void execute() throws Exception {
    Collection<Order> orders = getSelectedOrders(); // (2)
    Invoice invoice = Invoice.createFromOrders(orders); // (3)
    addMessage( // (4)
      "invoice_created_from_orders", invoice, orders);
  }

  private Collection<Order> getSelectedOrders() // (5)
    throws FinderException
  {
    Collection<Order> result = new ArrayList<Order>();
    for (Map key: getTab().getSelectedKeys()) { // (6)
      Order order = (Order)
        MapFacade.findEntity("Order", key); // (7)
      result.add(order);
    }
    return result;
  }
}

```

Realmente sencillo. Obtenemos la lista de los pedidos marcados en la lista (2), llamamos al método estático `createFromOrders()` (3) de `Invoice` y mostramos un mensaje (4). En este caso también ponemos la lógica real en la clase del modelo, no en la acción. Dado que la lógica aplica a varios pedidos y crea una nueva factura, el lugar natural para ponerlo es en un método estático de la clase

Invoice.

El método `getSelectedOrders()` (5) devuelve una colección con las entidades `Order` marcadas por el usuario en la lista. Para hacerlo, el método usa `getTab()` (6), disponible en `TabBaseAction` (1), que devuelve un objeto `org.openxava.tab.Tab`. El objeto `Tab` te permite manejar los datos tabulares de la lista. En este caso usamos `getSelectedKeys()` (6) que devuelve una colección con las claves de las filas seleccionadas. Dado que esas claves están en formato `Map` usamos `MapFacade.findEntity()` (7) para convertirlas en entidades `Order`.

Acuérdete de añadir el texto del mensaje al fichero *Invoicing-messages\_en.properties* en la carpeta *i18n*. El listado 8.19 muestra un posible texto.

#### Listado 8.19 Mensaje de confirmación en *Invoicing-messages\_en.properties*

```
invoice_created_from_orders=Invoice {0} created from orders: {1}
```

Eso es todo para la acción. Veamos la pieza que falta, el método `createFromOrders()` de la entidad `Invoice`.

### 8.2.2 Lógica de negocio en el modelo sobre varias entidades

La lógica de negocio para crear una nueva `Invoice` a partir de varias entidades `Order` está en la capa del modelo, es decir, en las entidades, no en la acción. No podemos poner el método en la clase `Order`, porque el proceso se hace a partir de varios `Orders`, no de uno. No podemos usar un método de instancia en `Invoice` porque todavía no existe el objeto `Invoice`, de hecho lo que queremos es crearlo. Por lo tanto, vamos a crear un método de factoría estático en la clase `Invoice` para crear una nueva `Invoice` a partir de varios `Orders`. Puedes ver este método en el listado 8.20.

#### Listado 8.20 Método `createFromOrders()` en entidad `Invoice`

```
public class Invoice extends CommercialDocument {
    ...

    public static Invoice createFromOrders(Collection<Order> orders)
        throws ValidationException
    {
        Invoice invoice = null;
        for (Order order: orders) {
            if (invoice == null) { // La primera vez, el primer pedido
                order.createInvoice(); // Reutilizamos la lógica para
                                    // crear una factura a partir de un pedido
                invoice = order.getInvoice(); // y cogemos la factura recién creada
            }
            else { // Para el resto de los pedido la factura ya está creada

```

```

        order.setInvoice(invoice); // Asigna la factura
        order.copyDetailsToInvoice(invoice); // Copia la línea. El método
    } // copyDetailsToInvoice es privado en Order.
    // por tanto tenemos que cambiarlo a público
    if (invoice == null) { // Si no hay pedidos
        throw new ValidationException(
            "impossible_create_invoice_orders_not_specified");
    }
    return invoice;
}
}

```

Usamos el primer Order para crear una nueva Invoice usando el método ya existente `createInvoice()` de Order. Entonces copiamos las líneas de los Orders restantes a la nueva Invoice. Además, asignamos la nueva Invoice como la Invoice de los Orders de la colección.

Si invoice es nulo al final del proceso, es porque la colección orders está vacía. En este caso lanzamos una `ValidationException`, ya que la acción no atrapa las excepciones, OpenXava muestra el mensaje de la `ValidationException` al usuario. Esto está bien. Si el usuario no marca los pedido y pulsa en el botón para crear la factura, le aparecerá este mensaje de error.

Usamos el método `copyDetailsToInvoice()` de Order. Este método era privado, por tanto necesitamos cambiarlo a público para poder usarlo desde Invoice. Observa el cambio en el listado 8.21.

#### Listado 8.21 Refinamientos en `copyDetailsToInvoice()` de Order

```

public class Order extends CommercialDocument {
    ...

    public private // public en vez de private
    void copyDetailsToInvoice(Invoice invoice)
    throws Exception // throws Exception se quita. Ahora se lanza una excepción runtime
    {
        try { // Envolvemos todo el código del método con un try/catch
            for (Detail orderDetail: getDetails()) {
                Detail invoiceDetail = (Detail)
                    BeanUtils.cloneBean(orderDetail);
                invoiceDetail.setOid(null);
                invoiceDetail.setParent(invoice);
                XPersistence.getManager()
                    .persist(invoiceDetail);
            }
        }
        catch (Exception ex) { // Así convertimos cualquier excepción
            throw new SystemException( // en una excepción runtime
                "impossible_copy_details_to_invoice", ex);
        }
    }
}

```



```
}
```

Además de cambiar 'private' por 'public' envolvemos cualquier excepción en una excepción *runtime*, de esta manera observamos la ya mencionada convención de usar excepciones *runtime* para los problemas inesperados.

Acuérdate de añadir los textos para los mensajes en el archivo *Invoicing-messages\_en.properties* de la carpeta *il8n*. El listado 8.22 muestra unos textos posibles.

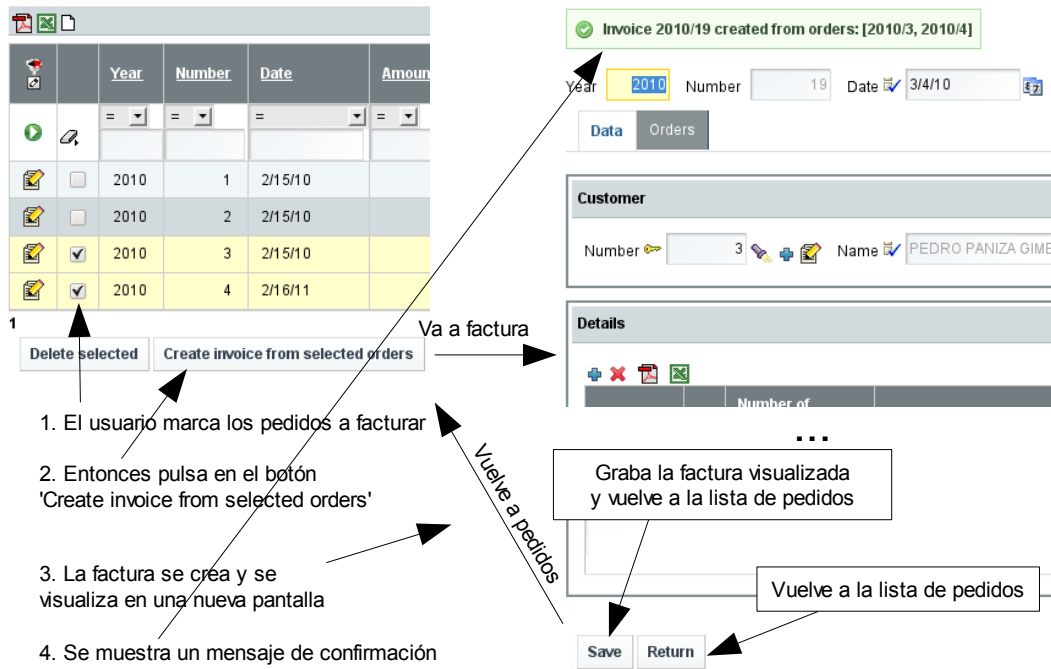
#### Listado 8.22 Error de validación en *Invoicing-messages\_en.properties*

```
impossible_create_invoice_orders_not_specified=Impossible to create invoice:
orders not specified
impossible_copy_details_to_invoice=Impossible to copy details from order to
invoice
```

Este no es el único error con el que el usuario puede encontrarse. Todas las validaciones que hemos escrito para *Invoice* y *Order* hasta ahora se aplican automáticamente, por lo tanto el usuario ha de escoger pedidos ya entregados y sin factura. La validación del modelo impide que el usuario cree una factura desde pedidos no apropiados.

### 8.3 Cambiar de módulo

Sería útil para el usuario que después de crear la factura a partir de varios pedidos, pudiera ver y editar la factura recién creada. Una forma de conseguir este comportamiento es creando un módulo sólo para editar una factura, es decir sin modo lista y sin las típicas acciones CRUD. De esta forma podemos cambiar a este módulo después de crear la factura para editarla. La figura 8.3 muestra el comportamiento deseado.



**Figura 8.3** Editar la factura después de crearla a partir de varios pedidos

Veamos como implementar este comportamiento.

### 8.3.1 Uso de *IChangeModuleAction*

El primer paso es modificar `CreateInvoiceFromSelectedOrdersAction` para cambiar a otro módulo después de su ejecución. El listado 8.23 muestra la modificación.

#### Listado 8.23 Modificación en acción para cambiar a un nuevo módulo

```
public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction
    implements IChangeModuleAction // Para cambiar a otro módulo después de la ejecución
{
    public String getNextModule() {
        return "CurrentInvoiceEdition"; // Nombre de módulo como está definido en
                                         // application.xml
    }

    public boolean hasReinitNextModule() {
        return true; // Así el módulo se inicializa cada vez que cambiamos a él
    }

    ...
}
```

Como puedes ver, solo has de implementar `IChangeModuleAction`. Esto te

obliga a añadir los métodos `getNextModule()` que devuelve el nombre del módulo tal como está definido en *application.xml*, y `hasReinitNextModule()`. Devolvemos *true* de `hasReinitNextModule()` porque escribiremos una acción *on-init* (acción ejecutada cuando el módulo se inicializa) en el módulo *CurrentInvoiceEdition* para cargar la factura correcta en la vista, por tanto necesitamos iniciar el módulo cada vez que cambiamos a él.

Obviamente, esto no funcionará hasta que tengamos el módulo *CurrentInvoiceEdition* definido. Haremos esto en la siguiente sección.

### 8.3.2 Módulo de solo detalle

El objetivo del módulo *CurrentInvoiceEdition* es visualizar una única factura y dar la opción de editarla.

Para definirlo edita el archivo *application.xml* y añade la definición de módulo del listado 8.24

**Listado 8.24 Módulo de solo detalle para editar una factura en application.xml**

```
<module name="CurrentInvoiceEdition">
  <model name="Invoice"/>
  <controller name="CurrentInvoiceEdition"/>
  <mode-controller name="Void"/> <!-- Así el módulo tiene sólo modo de detalle -->
</module>
```

Dado que este módulo es para editar una *Invoice* particular, no tiene modo lista, sino sólo modo detalle. Usamos *Void* como *mode-controller* para conseguirlo.

Este módulo sólo permite al usuario cambiar la *Invoice*, grabar los cambios o volver al módulo original. Para hacerlo define un controlador con estas acciones llamado *CurrentInvoiceEdition*. Has de añadirlo a *controllers.xml*, tal como se muestra en el listado 8.25.

**Listado 8.25 Controlador para editar una factura en controllers.xml**

```
<controller name="CurrentInvoiceEdition">

  <action name="save"
    class="org.openxava.invoicing.actions.SaveInvoiceAction"
    keystroke="Control S"/>

  <action name="return"
    class="org.openxava.actions.ReturnPreviousModuleAction"/>

</controller>
```

Las dos acciones de este controlador representan los dos botones, 'Save' y 'Return' que viste en la anterior figura 8.3.

### 8.3.3 Volviendo al módulo que llamó

SaveInvoiceAction es un pequeño refinamiento de la estándar SaveAction de OpenXava. El listado 8.26 muestra su código.

**Listado 8.26 Acción que graba la factura y vuelve al módulo que llamó**

```
public class SaveInvoiceAction
    extends SaveAction // Acción estándar de OpenXava para grabar el contenido de la vista
    implements IChangeModuleAction // Para navegación entre módulos
{

    public String getNextModule() {
        return PREVIOUS_MODULE; // Vuelve al módulo que llamó, Order en este caso
    }

    public boolean hasReinitNextModule() {
        return false; // No queremos inicializar el módulo Order
    }

}
```

La acción extiende de SaveAction sin sobrescribir el método execute(). Por lo tanto su comportamiento es exactamente el mismo que el de la acción genérica de OpenXava para grabar los datos visualizados en la base de datos. Adicionalmente, indicamos que la acción tiene que volver al módulo que la llamó, el módulo Order en nuestro ejemplo, cuando termine.

De esta forma cuando el usuario pulsa en 'Save' los datos de la factura se graban y vuelve a la lista de pedidos, listo para continuar creando facturas desde pedidos.

Para volver al módulo que llama tenemos que usar siempre PREVIOUS\_MODULE. No uses el nombre del módulo, como muestra el listado 8.27.

**Listado 8.27 Nunca usar el nombre de módulo para volver al módulo que llamó**

```
public String getNextModule() { return PREVIOUS_MODULE; } // Bien
public String getNextModule() { return "Order"; } // Muy MAL
```

Si usas PREVIOUS\_MODULE tienes la ventaja de que puedes llamar a este módulo desde varios módulos de la aplicación, y éste sabrá a que módulo volver en cada caso. Pero más importante todavía es el hecho de que OpenXava usa una pila de llamadas a módulos para poder volver, por tanto si llamas a un módulo que te ha llamado se produce un problema de reentrada.

Para el botón 'Return' usamos ReturnPreviousModuleAction, una acción incluida en OpenXava que simplemente vuelve al módulo que llamó.

### 8.3.4 Objeto de sesión global y acción on-init

El código actual está todavía incompleto. Cuando el usuario genera la factura el módulo `CurrentInvoiceEdition` se activa, pero está vacío, no muestra la factura. Hemos de llenar la vista del nuevo módulo con la factura recién creada. Aprendamos como compartir datos entre módulos.

Una forma de compartir datos entre módulos es declarando un objeto de sesión de ámbito global. Esto se consigue añadiendo una entrada en `controllers.xml` como se muestra en el listado 8.28.

**Listado 8.28 Objeto de sesión con ámbito global definido en controllers.xml**

```
<controllers>

...

<object name="invoicing_currentInvoiceKey"
  class="java.util.Map"
  scope="global"/>
<!--
  name="invoicing_currentInvoiceKey": El nombre tiene que ser único
  class="java.util.Map": El tipo del objeto
  scope="global": Compartido por todos los módulos. Por defecto es "module"
-->

...
```

Un objeto de sesión es un objeto asociado a la sesión del usuario, por lo tanto vivirá mientras que la sesión del usuario esté viva, y cada usuario tiene su propia copia del objeto. Si usas `scope="global"` el mismo objeto se compartirá por todos los módulos, en caso contrario cada módulo tiene su propia copia del objeto.

Declaramos el ámbito del objeto como global porque queremos usarlo para pasar datos desde el módulo `Order` al módulo `CurrentInvoiceEdition`. La forma de hacer esto es inyectándolo en la acción mediante la anotación `@Inject`<sup>12</sup>. Antes de llamar al método `execute()` de la acción, el objeto `invoicing_currentInvoiceKey` se inyecta en el campo `currentInvoiceKey` de la acción. Nota como el nombre del campo es el nombre del objeto de sesión sin el prefijo (sin `invoicing_` en este caso), aunque puedes inyectar el objeto en una propiedad con otro nombre si usas la anotación `@Named`. El listado 8.29 muestra el campo `currentInvoiceKey` con `@Inject` añadido a la acción.

**Listado 8.29 Campo `currentInvoiceKey` a ser inyectado del objeto de sesión**

```
...

import javax.inject.*;
```

<sup>12</sup> La anotación `@javax.inject.Inject` está definida por el estándar de Java JSR-330

```
public class CreateInvoiceFromSelectedOrdersAction ... {
    ...
    @Inject
    private Map currentInvoiceKey; // Un campo privado sin getter ni setter
    ...
}
```

Lo interesante de `@Inject` es que, además de inyectar el objeto en el campo antes de llamar a `execute()`, extrae el valor del campo y lo vuelve a poner en el contexto de la sesión después de ejecutar el método `execute()`. En otras palabras, si modificaras el valor del campo `currentInvoiceKey` de `CreateInvoiceFromSelectedOrdersAction` entonces el objeto de sesión `invoicing_currentInvoiceKey` se modificaría también. Por lo tanto, podemos usar esta acción para dar valor a este objeto de sesión. El listado 8.30 muestra la modificación en el código de la acción.

#### Listado 8.30 Llenar el objeto de sesión `currentInvoiceKey` desde la acción

```
public class CreateInvoiceFromSelectedOrdersAction ... {
    ...
    public void execute() throws Exception {
        Collection<Order> orders = getSelectedOrders();
        Invoice invoice = Invoice.createFromOrders(orders);
        addMessage("invoice_created_from_orders",
            invoice, orders);
        currentInvoiceKey = toKey(invoice); // Pone la clave de la recién creada
        // factura en el campo currentInvoiceKey, por lo tanto también
        // en el objeto de sesión invoicing_currentInvoiceKey
    }
    private Map toKey(Invoice invoice) { // Extrae la clave de la factura en formato mapa
        Map key = new HashMap();
        key.put("oid", invoice.getOid());
        return key;
    }
    ...
}
```

Después de la creación de la factura, ponemos la clave de la factura en el objeto de sesión. Dar valor a un objeto de sesión es pan comido, solo has de asignar un valor al campo declarado con `@Inject`. En este caso asignar valor a `setCurrentInvoiceKey()` es suficiente para llenar el objeto correspondiente `invoicing_currentInvoiceKey`. Después puedes usar este objeto desde otras acciones, ya que su ámbito es global, también desde las acciones de otros módulos.

## 179 Lección 8: Comportamiento y lógica de negocio

Vamos a crear una nueva acción en el módulo `CurrentInvoiceEdition` para cargar el valor de la factura creada en el módulo `Order` con `CreateInvoiceFromSelectedOrdersAction`. El listado 8.31 muestra la declaración de esta acción `load` en el archivo `controllers.xml`.

**Listado 8.31** La declaración de la acción `load` en `controllers.xml` con `on-init=true`

```
<controller name="CurrentInvoiceEdition">

    <action name="load"
        class="org.openxava.invoicing.actions.LoadCurrentInvoiceAction"
        hidden="true"
        on-init="true"/>
    <!--
        hidden="true": No hay un vínculo o botón en la pantalla para esta acción
        on-init="true": Se ejecuta automáticamente cuando el módulo se inicializa
    -->

    ...

</controller>
```

Declaramos la acción como `hidden=true`, así no será visible, y por tanto el usuario no tendrá la posibilidad de ejecutarla. Además, la declaramos como `on-init=true`, por tanto se ejecutará automáticamente cuando el módulo se inicialice.

Recuerda que llamamos a este módulo devolviendo `true` para `hasReinitNextModule()`, así `CurrentInvoiceEdition` se inicializa cada vez que se llama desde el módulo `Order`, por ende la acción `load` se llama siempre. Esta acción `load` es el lugar ideal para rellenar la vista con la factura recién creada. Veamos su código en el listado 8.32.

**Listado 8.32** Acción para cargar la última factura cargada en la vista

```
public class LoadCurrentInvoiceAction
    extends SearchByKeyAction { // Para llenar la vista a partir de la clave

    @Inject
    private Map currentInvoiceKey; // Para coger el valor del objeto de sesión
                                   // invoicing_currentInvoiceKey, llenado en el módulo Order
    public void execute() throws Exception {
        getView().setValues(currentInvoiceKey); // Pone la clave en la vista
        super.execute(); // Llena toda la vista a partir de los campos clave
    }

}
```

Extiende de `SearchByKeyAction` la cual es la acción estándar de OpenXava para buscar. `SearchByKeyAction` coge los campos clave de la vista, busca la entidad correspondiente, y rellena el resto de la vista a partir de la entidad. Por lo tanto, nosotros sólo hemos de llenar la vista con los valores de la clave antes de llamar a `super.execute()`.

Puedes ver como usando `currentInvoiceKey` accedemos a los valores de la clave almacenados ahí por `CreateInvoiceFromSelectedOrdersAction`. Has visto como usar un objeto de sesión para compartir datos entre acciones, incluso si éstas son de módulos diferentes.

El trabajo está casi terminado. Puedes probar el módulo `Order`: escoge varios pedidos y pulsa en el botón 'Create invoice from selected orders'. Entonces verás la factura creada en un módulo de solo detalle. Tal como viste en la anterior figura 8.3.

## **8.4 Pruebas JUnit**

El código que hemos escrito en esta lección no estará completo hasta que no escribamos las pruebas. Recuerda, todo código nuevo tiene que tener su correspondiente código de prueba. Escribamos pues las pruebas para estas dos nuevas acciones.

### **8.4.1 Probar la acción de modo detalle**

Primero probaremos la acción `Order.createInvoice`, la acción para crear una factura a partir del modo detalle del pedido visualizado. Reimprimimos aquí la figura 8.1 que muestra como funciona este proceso.



## 181 Lección 8: Comportamiento y lógica de negocio

Estamos en el módulo Order

1. El usuario escoge un pedido

2. Entonces pulsa en el botón 'Create invoice'

3. Se muestra un mensaje confirmando la creación de la nueva factura

4. Se muestra la nueva factura en la pestaña 'Invoice' del pedido

Las líneas de detalle de la nueva factura han sido copiadas del pedido actual

	Number of Product	Description of Product	Quantity	Price per unit	Amount
	1	Peopleware: Productive Projects and Teams	2	31.00	62.00
	2	Arco iris de lágrimas	1	15.00	15.00

**Figura 8.1 (reimpresión) Crear una factura desde un pedido usando una acción**

Ahora vamos a escribir un test para verificar que funciona justo de esta forma. Añade el método `testCreateInvoiceFromOrder()` del listado 8.33 a la clase `OrderTest`.

### Listado 8.33 El método `testCreateInvoiceFromOrder()` en `OrderTest`

```
public void testCreateInvoiceFromOrder() throws Exception {
    // Buscar el pedido
    searchOrderSusceptibleToBeInvoiced(); // Busca un pedido
    assertValue("delivered", "true"); // El pedido está entregado
    int orderDetailsCount = getCollectionRowCount("details"); // Toma nota de
    // la cantidad de detalles en el pedido
    execute("Sections.change", "activeSection=1"); // La sección de la factura
    assertValue("invoice.year", ""); // Todavía no hay factura
    assertValue("invoice.number", ""); // en este pedido

    // Crear la factura
    execute("Order.createInvoice"); // Ejecuta la acción que estamos probando (1)
    String invoiceYear = getValue("invoice.year"); // Verifica que ahora
```

```

assertTrue("Invoice year must have value", // hay una factura
    !Is.emptyString(invoiceYear)); // en la pestaña de factura (2)
String invoiceNumber = getValue("invoice.number");
assertTrue("Invoice number must have value",
    !Is.emptyString(invoiceNumber)); // Is.emptyString() es de org.openxava.util
assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
    " created from current order"); // El mensaje de confirmación (3)
assertCollectionRowCount("invoice.details", // La factura recién creada
    orderDetailsCount); // tiene el mismo número de detalles que el pedido (4)

// Restaurar el pedido para poder ejecutar la prueba la siguiente vez
setValue("invoice.year", "");
assertValue("invoice.number", "");
assertCollectionRowCount("invoice.details", 0);
execute("CRUD.save");
assertNoErrors();
}

```

Esta prueba pulsa el botón para ejecutar la acción `Order.createInvoice` (1), entonces verifica que una factura ha sido creada, está siendo visualizada en la pestaña de factura (2) y tiene la misma cantidad de líneas de detalle que el pedido actual (4). También verifica que se ha generado el mensaje de confirmación correcto (3).

Para ejecutarla es necesario escoger un pedido susceptible de ser facturado. Esto se hace en el método `searchOrderSusceptibleToBeInvoiced()` que vamos a examinar en la siguiente sección.

#### 8.4.2 Buscar una entidad para la prueba usando el modo lista y JPA

Para seleccionar un pedido adecuado para nuestra prueba usaremos JPA para determinar el año y número de ese pedido, y entonces usaremos el modo lista para seleccionar este pedido y editarlo en modo detalle. El listado 8.34 muestra los métodos para implementar esto.

##### Listado 8.34 Métodos de `OrderTest` para buscar un pedido usando JPA en lista

```

private void searchOrderSusceptibleToBeInvoiced() throws Exception {
    searchOrderUsingList("o.delivered = true and o.invoice = null"); // Envía
} // la condición, en este caso buscamos por un pedido entregado y sin factura

private void searchOrderUsingList(String condition) throws Exception {
    Order order = findOrder(condition); // Busca el pedido con la condición usando JPA
    String year = String.valueOf(order.getYear());
    String number = String.valueOf(order.getNumber());
    setConditionValues(new String [] { year, number }); // Llena el año y el número
    execute("List.filter"); // y pulsa en el botón filtrar en la lista
    assertListRowCount(1); // Sólo una fila, correspondiente al pedido buscado
    execute("Mode.detailAndFirst"); // Para ver el pedido en modo detalle
    assertValue("year", year); // Verifica que el pedido editado
    assertValue("number", number); // es el deseado
}

```

```

private Order findOrder(String condition) {
    Query query = XPersistence.getManager().createQuery( // Crea una consulta JPA
        "from Order o where o.deleted = false and " // a partir de la condición. Fíjate en
        + condition); // deleted = false para excluir los pedidos borrados
    List orders = query.getResultList();
    if (orders.isEmpty()) { // Es necesario al menos un pedido con la condición
        fail("To run this test you must have some order with " + condition);
    }
    return (Order) orders.get(0);
}

```

El método `searchOrderSusceptibleToBeInvoiced()` simplemente llama a un método más genérico, `searchOrderUsingList()`, para buscar una entidad por una condición. El método `searchOrderUsingList()` obtiene la entidad `Order` mediante `findOrder()`, entonces usa la lista para filtrar por el año y el número a partir de este `Order`, yendo a modo detalle al finalizar. El método `findOrder()` usa JPA simple y llano para buscar.

Como puedes ver, combinar el modo lista con JPA es una herramienta muy útil en ciertas circunstancias. Usaremos los métodos `searchOrderUsingList()` y `findOrder()` en las siguientes pruebas.

#### 8.4.3 Probar que la acción se oculta cuando no aplica

Recuerda que refinamos el módulo `Order` en la sección 8.1.7 para que mostrara la acción para crear la factura solo cuando el pedido visualizado fuese susceptible de ser facturado. El listado 8.35 muestra el método de prueba para este caso.

##### Listado 8.35 Probar que la acción se oculta correctamente en `OrderTest`

```

public void testHidesCreateInvoiceFromOrderWhenNotApplicable()
    throws Exception
{
    searchOrderUsingList(
        "delivered = true and invoice <> null"); // Si el pedido ya tiene factura
    assertNoAction("Order.createInvoice"); // no se puede facturar otra vez

    execute("Mode.list");

    searchOrderUsingList(
        "delivered = false and invoice = null"); // Si el pedido no está entregado
    assertNoAction("Order.createInvoice"); // no se puede facturar

    execute("CRUD.new"); // Si el pedido todavía no está grabado
    assertNoAction("Order.createInvoice"); // no puede ser facturado
}

```

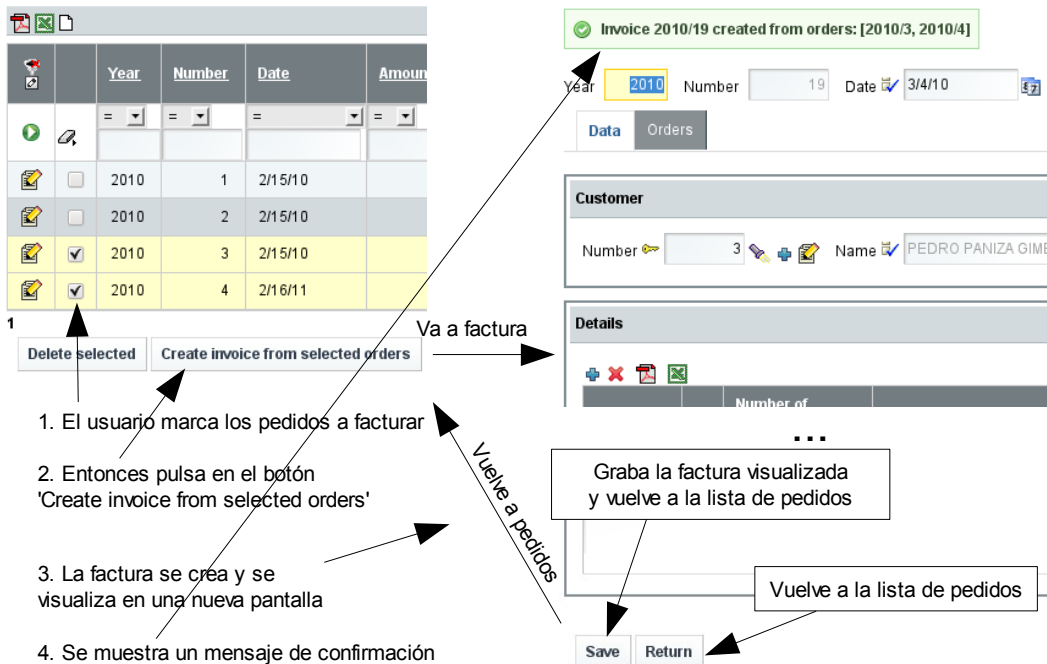
Probamos tres casos en los que el botón para crear la factura no tiene que estar presente. Fíjate en el uso de `assertNoAction()` para preguntar si el vínculo o botón para una acción está presente en la interfaz de usuario. Aquí estamos

reutilizando el método `searchOrderUsingList()` desarrollado en la sección anterior.

Ya hemos probado que el botón está presente cuando el pedido es adecuado en la prueba `testCreateInvoiceFromOrder()`, porque `execute()` falla si la acción no está en la interfaz de usuario.

#### 8.4.4 Probar la acción de modo lista

Ahora probaremos `Order.createInvoiceFromSelectedOrders`, la acción que crea una factura desde varios pedidos en modo lista (figura 8.3).



**Figura 8.3 (reimp.) Editar la factura después de crearla a partir de varios pedidos**

Escribamos una prueba para verificar que esto funciona justo de esta forma. Añade el método `testCreateInvoiceFromSelectedOrders()` del listado 8.36 a la clase `OrderTest`.

#### Listado 8.36 El método `testCreateInvoiceSelectedOrders()` en `OrderTest`

```
public void testCreateInvoiceFromSelectedOrders() throws Exception {
    assertOrder(2010, 9, 2, 362); // El pedido 2010/9 tiene 2 líneas y 362 de importe base
    assertOrder(2010, 10, 1, 126); // El pedido 2010/10 tiene 1 línea y 126 de importe base

    execute("List.orderBy", "property=number"); // Ordena la lista por número
    checkRow( // Marca la fila a partir del número de fila
        getDocumentRowInList("2010", "9") // Obtiene la fila del año y número del pedido
    ); // por tanto, esta línea marca la línea del pedido 2010/9 en la lista (1)
    checkRow(
```

```

        getDocumentRowInList("2010", "10")
    ); // Marca el pedido 2010/10 en la lista (1)

    execute("Order.createInvoiceFromSelectedOrders"); // Ejecuta la acción que
                                                    // estamos probando (2)

    String invoiceYear = getValue("year"); // Ahora estamos viendo el detalle de
    String invoiceNumber = getValue("number"); // la factura recién creada
    assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
        " created from orders: [2010/9, 2010/10]"); // El mensaje de confirmación
    assertCollectionRowCount("details", 3); // Confirma que el número de líneas de la
        // factura recién creada es la suma de la de los pedidos fuente (3)
    assertValue("baseAmount", "488.00"); // Confirma que el importe base de la factura
        // recién creada es la suma de la de los pedidos fuente (4)
    execute("Sections.change", "activeSection=1"); // Cambia a la pestaña de
        // pedidos de la factura
    assertCollectionRowCount("orders", 2); // La nueva factura tiene 2 pedidos (5)
    assertValueInCollection("orders", 0, 0, "2010"); // y son los correctos
    assertValueInCollection("orders", 0, 1, "9");
    assertValueInCollection("orders", 1, 0, "2010");
    assertValueInCollection("orders", 1, 1, "10");

    assertAction("CurrentInvoiceEdition.save"); // Los botones 'Save' (6)
    assertAction("CurrentInvoiceEdition.return"); // y 'Return' (6)

    checkRowCollection("orders", 0); // Seleccionamos los 2 pedidos
    checkRowCollection("orders", 1);
    execute("Collection.removeSelected", // y los borramos, para ejecutar esta prueba
        "viewObject=xava_view_section1_orders"); // otra vez usando los mismo pedidos
    assertNoErrors();

    execute("CurrentInvoiceEdition.return"); // Vuelve a la lista de pedidos (7)
    assertDocumentInList("2010", "9"); // Confirma que estamos realmente
    assertDocumentInList("2010", "10"); // en la lista de pedidos
}

```

Esta prueba marca dos pedidos (1) y pulsa en el botón 'Create invoice from selected orders' (2). Entonces verifica que se ha creado una nueva factura con el número correcto de líneas (3), importe base (4) y lista de pedidos (5). También verifica que las acciones 'Save' y 'Return' están disponibles (6) y usa el botón 'Return' para volver a la lista de pedidos (7).

Usamos `getDocumentRowInList()` y `assertDocumentInList()`, métodos de la clase base `CommercialDocumentTest`, que fueron definidos originalmente como privados, por lo tanto tenemos que redefinirlos como protegidos para poder utilizarlos desde `OrderTest`. Edita `CommercialDocumentTest` y haz los cambios del listado 8.37.

#### Listado 8.37 Cambia de private a protected en CommercialDocumentTest

```

protected private void assertDocumentInList(String year, String number) ...

protected private int getDocumentRowInList(String year, String number) ...

```

El único detalle pendiente es el método `assertOrder()` que veremos en la

siguiente sección.

#### 8.4.5 Verificar datos de prueba

En la lección 6 aprendiste como confiar en datos existentes en la base de datos para tus pruebas. Obviamente, si tu base de datos se altera accidentalmente tus pruebas, aunque correctas, no pasaran. Por tanto, verificar los valores de la base de datos antes de ejecutar la prueba que confía en ellos es una buena práctica. En nuestro ejemplo lo hacemos llamando a `assertOrder()` al principio. Veamos el contenido de `assertOrder()` en el listado 8.38.

**Listado 8.38 Método para verificar el estado de un pedido ya existente**

```
private void assertOrder(
    int year, int number, int detailsCount, int baseAmount)
{
    Order order = findOrder("year = " + year + " and number=" + number);
    assertEquals("To run this test the order " +
        order + " must have " + detailsCount + " details",
        detailsCount, order.getDetails().size());
    assertTrue("To run this test the order " +
        order + " must have " + baseAmount + " as base amount",
        order.getBaseAmount().compareTo(new BigDecimal(baseAmount)) == 0);
}
```

Este método busca un pedido y verifica la cantidad de líneas y el importe base. Usar este método tiene la ventaja de que si los pedidos necesarios para la prueba no están en la base de datos con los valores correctos obtienes un mensaje preciso. Así, no derrocharás tu tiempo intentando adivinar que es lo que está mal. Esto es especialmente útil si la prueba no la está ejecutando el programador original.

#### 8.4.6 Probar casos excepcionales

Dado que la acción para crear la factura se oculta si el pedido no está listo para ser facturado, no podemos probar el código para los casos excepcionales que escribimos en la sección 8.1.5 desde modo detalle. Sin embargo, en modo lista el usuario todavía tiene la opción de escoger cualquier pedido para facturar. Por tanto, intentaremos crear la factura desde la lista de pedidos para probar que los casos excepcionales se comportan correctamente. El listado 8.39 muestra el código de prueba en `OrderTest`.

**Listado 8.39 Probar casos excepcionales creando una factura desde un pedido**

```
public void testCreateInvoiceFromOrderExceptions() throws Exception {
    assertCreateInvoiceFromOrderException( // Verifica que cuando el pedido ya tiene (1)
        "delivered = true and invoice <> null", // factura se produce el error correcto
        "Impossible to create invoice: the order already has an invoice"
    );
}
```

```

    assertCreateInvoiceFromOrderException( // Verifica que cuando el pedido no está (2)
        "delivered = false and invoice = null", // entregado se produce el error correcto
        "Impossible to create invoice: the order is not delivered yet"
    );
}

private void assertCreateInvoiceFromOrderException(
    String condition, String message) throws Exception
{
    Order order = findOrder(condition); // Busca el pedido por la condición (3)
    int row = getDocumentRowInList( // y obtiene el número de fila para ese pedido (4)
        String.valueOf(order.getYear()),
        String.valueOf(order.getNumber())
    );
    checkRow(row); // Marca la fila (5)
    execute("Order.createInvoiceFromSelectedOrders"); ← Trata de crear la factura (6)
    assertError(message); // ¿Se ha mostrado el mensaje esperado? (7)
    uncheckRow(row); // Desmarca la fila, así podemos llamar a este método otra vez
}

```

La prueba verifica que el mensaje es el correcto cuando tratamos de crear una factura a partir de un pedido que ya tiene factura (1), y también desde un pedido no entregado todavía (2). Para hacer estas verificaciones llama al método `assertCreateInvoiceFromOrderException()`. Este método busca la entidad `Order` usando la condición (3), localiza la fila donde la entidad se está visualizando (4) y la marca (5). Después, la prueba ejecuta la acción (6) y verifica que el mensaje esperado se muestra (7).

## 8.5 Resumen

La sal de tu aplicación son las acciones y los métodos. Gracias a ellos puedes convertir una simple aplicación de gestión de datos en una herramienta útil. En este caso, por ejemplo, hemos provisto al usuario con una forma de crear automáticamente facturas desde pedidos.

Has aprendido como crear métodos de lógica de negocio tanto estáticos como de instancia, y como llamarlos desde acciones de modo detalle y modo lista. Por el camino has visto como ocultar y mostrar acciones, usar excepciones, validar en las acciones, cambiar a otro módulo y cómo hacer las pruebas automáticas de todo esto.

Todavía nos quedan muchas cosas interesante por aprender, por ejemplo en la siguiente lección vamos a refinar el comportamiento de las referencias y colecciones.